Distributed Systems
COMP90015 2019 SM1
File Systems
Lectures by Aaron Harwood
© University of Melbourne 2019

Summary

- file service architecture
- network file system
- enhancements and further developments

A basic *distributed file system* emulates the same functionality as a (non-distributed) file system for client programs running on multiple remote computers.

Advanced distributed file systems go much further by e.g. maintaining replica files and provide bandwidth and timing guarantees for multimedia data streaming.
A file system provides a convenient programming interface for disk storage along with features such as access control and file-locking that allows file sharing.

A *file service* allows programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet.

Hosts that provide a file service can be optimized for persistent storage devices, e.g. for multiple disk drives, and can supply file services for a wide range of other services in an organization, e.g. for the web services and email services. This further facilitates management of the persistent storage, including backups and archiving.

Without using a distributed file system, a user must "manually" copy files from one machine to another, either using a network copy utility, by email, or by some other means like removable media.

Characteristics of file systems

Files contains both *data* and *attributes*. Data is usually in the form of a sequence of bytes and can be accessed and modified. Attributes include things like the length of the file, timestamps, file type, owner's identity and access-control lists.

Files have a name. Some files are directories that contain a list of other files; and they may themselves be (sub-) directories. This leads to a hierarchical naming scheme for the file. The *pathname* is the concatenation of the directory names and the file name.

Typical layers of a file system include:

- Directory module: relates file names to IDs
- File module: relates file IDs to particular files
- Access control module: checks permission for operation requested

- File access module: reads or writes file data or attributes
- Block module: access and allocates disk blocks
- Device module: disk I/O and buffering

UNIX file system operations are shown below:

- `fdes=open(name,mode)` - Opens an existing file with the given name
- `fdes=creat(name,mode)` - Creates a new file with the given name
- `status=close(fdes)` - Closes the open file
- `cnt=read(fdes,buf,n)` - Transfers bytes from the file into the buffer
- `cnt=write(fdes,buf,n)` - Transfers bytes from the buffer into the file
- `pos=lseek(fdes,offs,whence)` - Moves the read-write pointer to a new position in the file
- `status=unlink(name)` - Removes the file name from the directory structure
- `status=link(name1,name2)` - Adds a new name for the first named file
- `status=stat(name,buf)` - Gets the file attributes for the file

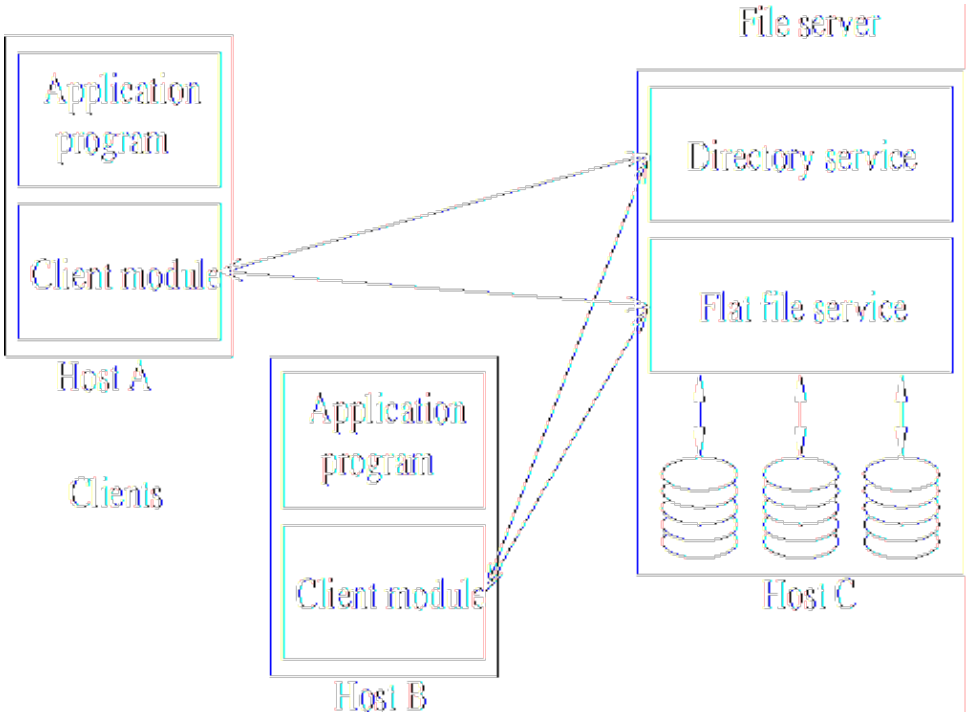
Distributed file system requirements

- **Transparency**:
  - *Access transparency* -- Client programs should be unaware of the distribution of files. Same API is used for accessing local and remote files and so programs written to operate on local files can, unchanged, operate on remote files.
  - *Location transparency* -- Client programs should see a uniform file name space; the names of files should be consistent regardless of where the files are actually stored and where the clients are accessing them from.
  - *Mobility transparency* -- Client programs and client administration services do not need to change when the files are moved from one place to another.
  - *Performance transparency* -- Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.
  - *Scaling transparency* -- The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.
- **Concurrent file updates**: Multiple clients' updates to files should not interfere with each other. Policies should be manageable.

- **File replication**: Each file can have multiple copies distributed over several servers, that provides better capacity for accessing the file and better fault tolerance.
- **Hardware and operating system heterogeneity**: The service should not require the client or server to have specific hardware or operating system dependencies.
- **Fault tolerance**: Transient communication problems should not lead to file corruption. Servers can use at-most-once invocation semantics or the simpler at-least-once semantics with *idempotent* operations. Servers can also be *stateless*.
- **Consistency**: Multiple, possibly concurrent, access to a file should see a consistent representation of that file, i.e. differences in the files location or update latencies should not lead to the file looking different at different times. File meta data should be consistently represented on all clients.
- **Security**: Client requests should be authenticated and data transfer should be encrypted.
- **Efficiency**: Should be of a comparable level of performance to conventional file systems.

File service architecture

- *Flat file service*: The flat file service is concerned with implementing operations on the contents of files. A *unique file identifier* (UFID) is given to the flat file service to refer to the file to be operated on. The UFID is unique over all the files in the distributed system. The flat file service creates a new UFID for each new file that it creates.
- *Directory service*: The directory service provides a mapping between text names and their UFIDs. The directory service creates directories and can add and delete files from the directories. The directory service is itself a client of the flat file service since the directory files are stored there.
- *Client module*: The client module integrates the directory service and flat file service to provide whatever application programming interface is expected by the application programs. The client module maintains a list of available file servers. It can also cache data in order to improve performance.



Though the client module is shown as directly supporting application programs, in practice it integrates into a virtual file system.

Flat file service interface

The flat file service interface is shown in the next slide, in terms of RPC calls.

Recall that the UNIX interface shown earlier requires that the UNIX file system maintains state, i.e. a file pointer, that is manipulated during reads and writes.

The flat file service interface differs from the UNIX interface mainly for reasons of fault tolerance:

- *repeatable operations* -- with the exception of `Create()`, the operations are idempotent, allowing the use of at-least-once RPC semantics.

- *stateless server* -- the flat file service does not need to maintain any state and can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

Also note that UNIX files require an explicit open command before they can be accessed, while files in the flat file service can be accessed immediately.

- `Read(UFID, i, n) -> Data` - Reads up to n items from position i in the file
- `Write(UFID, i, Data)` - Writes the data starting at position i in the file. The file is extended if necessary
- `Create() -> UFID` - Creates a new file of length 0 and returns a UFID for it
- `Delete(UFID)` - Removes the file from the file store
- `GetAttributes(UFID) -> Attr` - Returns the file attributes for the file
- `SetAttributes(UFID,Attr)` - Sets the file attributes.

Flat file service access control

The service needs to authenticate the RPC caller and needs to ensure that illegal operations are not performed, e.g. that UFIDs are legal and that files access privileges are not ignored.

The server cannot store any access control state as this would break the idempotent property.

- An access check can be made whenever a file name is converted to a UFID and the results can be encoded in the form of a capability that is returned to the client for submission to the flat file server.
- A user identity can be submitted with every client request, and access checks can be performed by the flat file server for every file operation.

Directory service interface

The primary purpose of the directory service is to provide a translation from file names to UFIDs. An abstract directory service interface is shown in the next slide.

The directory server maintains directory files that contain mappings between text file names and UFIDs. The directory files are stored in the flat file server and so the directory server is itself a client to the flat file server.

A hierarchical file system can be built up from repeated accesses. E.g., the root directory has name "/" and the contains subdirectories with names "usr", "home", "etc", which themselves contain other subdirectories or files. A client function can make requests for the UFIDs in turn, to proceed through the path to the file or directory at the end.

- `Lookup(Dir, Name) -> UFID` - Returns the UFID for the file name in the given directory
- `AddName(Dir, Name, UFID)` - Adds the file name with UFID to the directory
- `UnName(Dir, Name)` - Remove the file name from the directory
- `GetNames(Dir, Pattern) -> Names` - Returns all the names in the directory that match the pattern.

File group

A *file group* is a collection of files located on a given server. A server may hold several file groups and file groups can be moved between servers, but a file cannot change file group.

File groups allow the file service to be implemented over several servers.

Files are given UFIDs that ensure uniqueness across different servers, e.g. by concatenating the server IP address (32 bits) with a date that the file was created (16 bits). This allows the files in a group, i.e. that have a common part to their UFID called the *file group identifier*, to be relocated to a different server without conflicting with files already on that server.

The file service needs to maintain a mapping of UFIDs to servers. This can be cached at the client module.

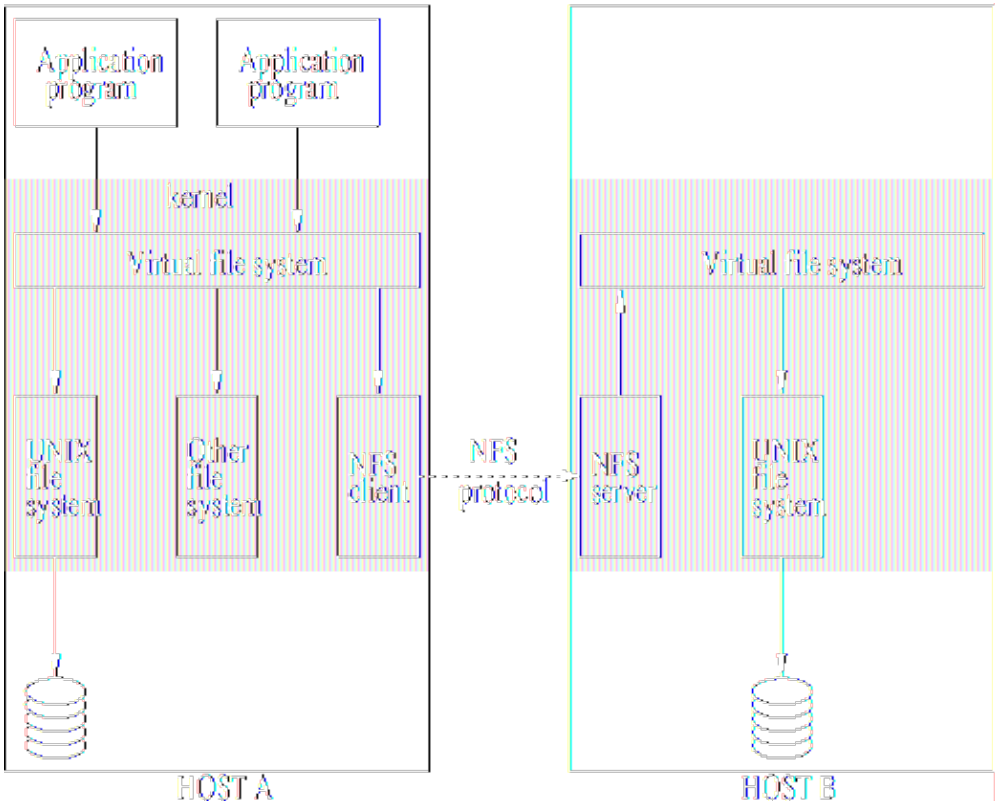
Sun Network File System

The Sun Network File System (NFS) follows the abstract system shown earlier.

There are many implementations of NFS and they all follow the NFS protocol using a set of RPCs that provide the means for the client to perform operations on the remote file store.

We consider a UNIX implementation.

The NFS client makes requests to the NFS server to access files.



Virtual file system

UNIX uses a *virtual file system* (VFS) to provide transparent access to any number of different file systems. The NFS is integrated in the same way.

The VFS maintains a VFS structure for each filesystem in use. The VFS structure relates a remote filesystem to the local filesystem; i.e. it combines the remote and local file system into a single filesystem.

The VFS maintains a *v-node* for each open file, and this records an indicator as to whether the file is local or remote.

If the file is local then the v-node contains a reference to the file's *i-node* on the UNIX file system.

If the file is remote then the v-node contains a reference to the file's NFS *file handle* which is a combination of *filesystem identifier*, i-node number and whatever else the NFS server needs to identify the file.

Client integration

The NFS client is integrated within the kernel so that:

- user programs can access files via UNIX system calls without recompilation or reloading;
- a single client module serves all of the user-level processes, with a shared cache;
- the encryption key used to authenticate user IDs passed to the server can be retained in the kernel, preventing impersonation by user-level clients.

The client transfers blocks of files from the server host to the local host and caches them, sharing the same buffer cache as used for local input-output system. Since several hosts may be accessing the same remote file, caching presents a problem of consistency.

Server interface

The NFS server interface integrates both the directory and file operations in a single service. The creation and insertion of file names in directories is performed by a single create operation, which takes the text name of the new file and file handle for the target directory as arguments.

The primitives of the interface largely resemble the UNIX filesystem primitives.

Mount service

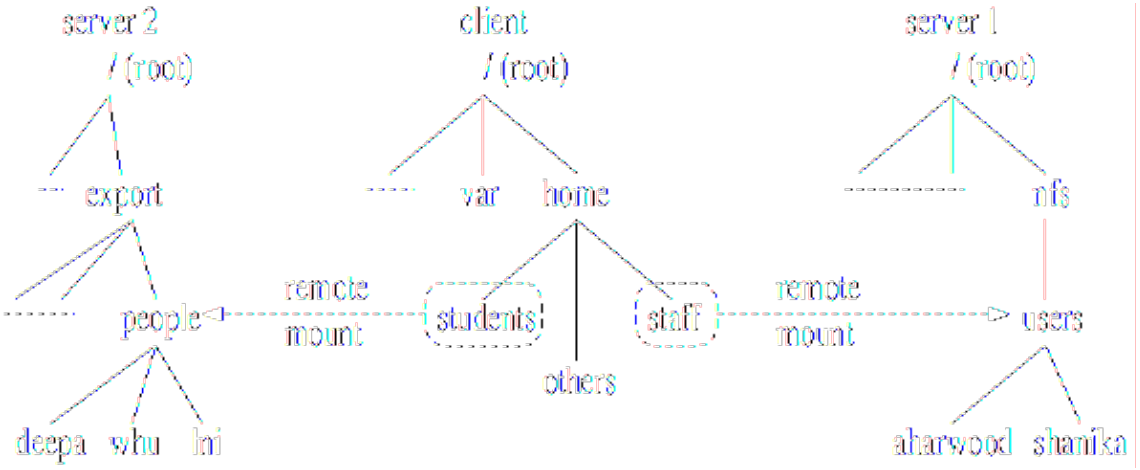
Each server maintains a file that describes which parts of the local filesystems that are available for remote mounting.

```
aharwood@htpc:~$ cat /etc/exports
# /etc/exports: the access control list for
#               filesystems which may be exported
#               to NFS clients.  See exports(5).
/store          192.168.1.0/255.255.255.0(rw)
```

In the above example all hosts on the subnet can mount the filesystem directory `/store` with read and write access.

A *hard-mounted* filesystem will block on each access until the access is complete. A *soft-mounted* filesystem will retry a few times and then return an error to the calling process.

Example NFS mounting



In the example `/etc/fstab`, the line starting `htpc:/store` states that the remote directory on `htpc` called `/store` should be mounted on the local filesystem; the name of the local mount point is given by the second parameter `/mnt/store`.

```
[aharwood@home ~]$ cat /etc/fstab
# This file is edited by fstab-sync - see 'man fstab-sync' for details
/dev/VolGroup00/LogVol00 /                       ext3    defaults        1 1
LABEL=/boot              /boot                   ext3    defaults        1 2
none                     /dev/pts                devpts  gid=5,mode=620  0 0
none                     /dev/shm                tmpfs   defaults        0 0
none                     /proc                   proc    defaults        0 0
none                     /sys                    sysfs   defaults        0 0
/dev/VolGroup00/LogVol01 swap                    swap    defaults        0 0
htpc:/store     /mnt/store    nfs   nfsvers=3,wsize=8192,rsize=8192,rw      0 0
/dev/hdc                 /media/cdrecorder       auto pamconsole,exec,noauto,managed 0 0
```

Server caching

In conventional UNIX systems, data read from the disk or pages are retained in a main memory buffer cache and are evicted when the buffer space is required for other pages. Accesses to cached data does not require a disk access.

Read-ahead anticipates read accesses and fetches the pages following those that have been recently read.

Delayed-write or write-back optimizes writes to the disk by only writing pages when both they have been modified and when they are evicted. A UNIX `sync` operation flushes modified pages to disk every 30 seconds.

This works for a conventional filesystem, on a single host, because there is only one cache and all file

accesses cannot bypass the cache.

Use of the cache at the server for client reads does not introduce any problems.

However use of the cache for writes requires special care to ensure that client can be confident that the writes are persistent, especially in the event of of a server crash.

There are two options for cache policies that are used by the server:

- *Write-through* -- data is written to cache and also directly to the disk. This increases disk I/O and increases the latency for write operations. The operation completes when the data has been written to disk.
- *Commit* -- data is written to cache and is written to disk when a commit operation is received for the data. A reply to the commit is sent when the data has been written to disk.

The first option is poor when the server receives a large number of write requests for the same data. It however saves network bandwidth.

The second option uses more network bandwidth and may lead to uncommitted data being lost. However it receives the full benefit of the cache.

Client caching

The NFS client also caches data reads, writes, attributes and directory operations in order to reduce network I/O.

Caching at the client introduces the cache consistency problem since now there is a cache at the client and the server, and there may be more than one client as well, each with its own cache.

Note that reading is a problem as well as writing, because a write on another client in between two read operations will lead to the second read operation being incorrect.

In NFS, clients poll the server to check for updates.

Let **T_c** be the time when a cache block was last validated by the client.

Let **T_m** be the time when a block was last modified.

A cache block is said to be valid at time **T** if (i) **T-T*c* < *t*** where **t** is a given _freshness interval, or (ii) the value of **T_m** at the client matches the value at the server:

(T-T*c* < *t*) *or* (*T*{m,client}=T_{m,server})

A small value for **t** leads to a close approximation of one-copy consistency, at the cost of greater network I/O.

In Sun Solaris clients **t** is set adaptively in the range 3 to 30 seconds depending on file update frequency. The range is 30 to 60 seconds for directories, since there is a lower risk of concurrent update.

The validity check is made on each access to cache block. If the first half of the check is true then the second half of the check need not be made. The first half of the check does not require network I/O.

A separate **T*{m,server}*** is kept by the server for the file attributes. If the first half of the check is found to false then the client contacts the server and retrieves **T{m,server}** for the file attributes. If it matches **T*{m,client}*** then the cache block is valid and the client sets **Tc** to the current time. If they do not match then the cache block is invalid and the client must request a new copy from the server.

Traffic can be reduced by applying new values of **T_{m,server}** to all relevant cache blocks and by piggy-backing attribute values on every file operation.

Write-back is used for writes, where modified files are flushed when a file is closed or when a sync operation takes place in the VFS. Special purpose daemons are used to do this asynchronously.

NFS summary

- *Access transparency*: Yes. Applications programs are usually not aware that files are remote and no changes are need to applications in order to access remote files.
- *Location transparency*: Not enforced. NFS does not enforce a global namespace since client filesystems may mount shared filesystems at different points. Thus an application that works on one client may not work on another.
- *Mobility transparency*: No. If the server changes then each client must be updated.
- *Scalability*: Good, could be better. The system can grow to accommodate more file servers as needed. Bottlenecks are seen when many processes access a single file.
- *File replication*: Not supported for updates. Additional services can be used to facilitate this.
- *Hardware and operating system heterogeneity*: Good. NFS is implemented on almost every known operating system and hardware platform.
- *Fault tolerance*: Acceptable. NFS is stateless and idempotent. Options exist for how to handle failures.

- *Consistency*: Tunable. NFS is not recommended for close synchronization between processes.
- *Security*: Kerberos is integrated with NFS. Secure RPC is also an option being developed.
- *Efficiency*: Acceptable. Many options exist for tuning NFS.